Enforcing Data Availability in Structured *Peer-to-Peer* Storage Systems *With Zero* Replica Migration

Mesaac Makpangou
REGAL Team
INRIA/LIP6 (UPMC)
Paris, France
Email: mesaac.makpangou@lip6.fr

***Abstract*—This paper presents a structured peer-to-peer storage substrate that exploits notifications issued by the underlying network maintenance layer to enforce data availability, while avoiding both application-level replica tracking and unneeded replica migrations. This system enforces a multiple keys replication approach. Each peer advertises its stored contents to a set of watchers picked from the set of peers within this peer's neighborhood. When a peer departs from the overlay network, its watchers initiate repairs of losses due to this departure. Thanks to the location of watchers within each watched peer's neighborhood, on peer departure, one can reduce the overall loss repair delay, and hence the probability of loosing for ever a stored content. The analytical evaluation shows that the proposed replica maintenance substrate generates far less overhead than a leaf set based replica maintenance system. Furthermore, on node arrivals, the overhead incurred by this proposal does not depend on the size of stored contents. This makes this substrate an interesting building block for peer-to-peer storage systems destined to store large-size objects.**

***Index Terms*—Peer-to-peer storage system, replica maintenance, flexible replication, distributed algorithms.**

I. Introduction

One challenge for a structured peer-to-peer storage system is to efficiently enforce stored contents availability in the face of node churn. One well known technique to address this issue is data replication. Number of existing structured storage systems enforce the leaf set (or successor) based replication approach. Each data item is replicated at its replica set (i.e., the k closest nodes to its root node, where k is the replication degree enforced by the system). In practice, upon each node departure, a replica maintenance procedure is run to create replicas that are lost due to this departure. Also when a node joins the overlay, it cooperates with its peers to determine replicas to migrate at the new coming node. While this solution to maintain data availability is simple, the overhead due to replica migrations increases with the number and the total size of stored contents [1]. This could jeopardize the overall performance of the system, especially if we consider the replication of large-size objects.

To enforce replica availability while avoiding unneeded replica migrations, one approach is to use multiple publication keys and to let the storage system computes the suitable number of storage keys, then uses them to place object replicas within the storage overlay network. This approach requires a means to efficiently track the availability of individual replicas. One solution is to check periodically the availability of each replica and to recreate a replica only when a replica is really lost. Such an active tracking of replicas has two drawbacks. Firstly, the loss repair delay is likely too high: to limit the tracking cost (i.e, number of messages and consumed bandwidth), one tends to enforce a large probe period; unfortunately, the larger the probe period, the higher the replica loss repair time and consequently the higher the risk to lose forever a stored data [2], [3]. Secondly, the active tracking doesn't scale: as the number of replicas augments, the tracking cost augments too; at some point, there will be not enough resource left for useful work.

This paper presents a Peer-to-peer Watching System (*Pws*), a replica maintenance substrate for structured storage utilities. *Pws* relies on a distributed hash table (DHT) abstraction; existing implementation of this abstraction as proposed in [4]–[7]. *Pws* enforces the multiple publication key replication approach. It associates with each storing node a *watch set*, a subset of nodes located within this node's neighborhood. Each storing node advertises to its watchers (i.e., members of its watch set) each object (i.e., replica or manager) that it stores, together with the identifier of the group of managers capable to handle this object's loss. Whenever a watcher detects that the node that it is watching departed (voluntarily or involuntarily) from the storage overlay network, it notifies manager groups of objects stored at that node to repair their losses. *Pws* is layered on top of an underlying network overlay that provides the common application programming interface specified in [8]. In particular, *Pws* assumes that the underlying network management system notifies each node whenever an event (i.e., node departure or arrival) that impacts this node's neighborhood occurs. *Pws* exploits these already existing notifications to track replica availability and to ensure that each node's watch set members are within this node neighborhood. Thanks to both the location of watchers within each node's neighborhood and notifications of changes in the neighborhood by the underlying network management layer, *Pws* detects

 ISBN: 978-1-61208-280-6

node departure (and hence stored contents loss) with (almost) none delay. This contributes to reduce the replica repair time which is a key metric that impacts the data availability in peer-to-peer storage systems [2].

The contribution of this paper is twofold. Firstly, a replica maintenance substrate that permits to build structured peer-to-peer storage systems that enforce data availability without incurring the cost of unneeded creations of replicas and of a periodic replica tracking mechanism, while enabling the detection of replica loss with almost no delay. Secondly, a flexible replication model that separates replication concerns: number of replicas, replica loss repair strategy, replica placement, replica watching. Applications control the replication scheme and the replica repair strategy, while the system is responsible of locating and watching alive replicas. This model permits providers to adapt the replica maintenance strategy, and hence the quality of service offered to end-users, on a per object basis. A detailed description of the main algorithms, together with the evaluation of the system overhead are provided.

The rest of the document is organized as follows. Section II presents related work, while Section III gives an overview of the *Pws* system. Then Section IV details the main algorithms and protocols of *Pws*. Section V evaluates *Pws* cost and compares it to a basic leaf set-based replica maintenance. Finally, Section VI draws some concluding remarks.

II. Related Work

Several neighborhood-based replication algorithms have been proposed to ensure data availability in structured peer-to-peer storage systems. For these systems, each data is stored at its root node and at a subset of neighbors. For instance, in PAST [9], a large-scale persistent storage utility using Pastry [6], the replicas of a file are stored in the k nodes that are numerically closest to the file identifier, where k is the replication degree to enforce [10]. One main advantage of the neighborhood-based replication is its capacity to efficiently tolerate node failure. When a node fails, a node in the neighborhood of the failed node is automatically promoted the responsible of data that were stored at that node and will be the target of lookup requests. While this solution to maintain data availability is simple and transparent to applications, the overhead incurred to create new replicas on replica set changes is unacceptable for storage systems that experience high churn. RelaxDHT [11] proposes to relax the replica placement constraint such as to avoid creating replicas when this is not mandatory to preserve data availability. For each stored object, replicas can be anywhere in the leaf-set (i.e, not necessary at the k closest nodes to the root node); however, the root node maintains meta-data describing its replica set. The root node periodically, sends messages to the replica set peers so that they keep storing their replicas. While this relaxation constitutes a real improvement, it still incurs unneeded replica creations when a member of a replica-set is put out of the root node's neighborhood, following node joins. *Pws* extends this relaxation and enforces a complete separation of a number of concerns that are often tightly coupled: replica placement, replica location, and replica maintenance. Thanks to this separation, *Pws* proposes a scalable way to track replicas of each object scattered over the network, together with a flexible replication management that permits to adapt the replication strategy (number of replicas, placements) according to the quality of service required by the object provider.

A. Ghdsi et al. [12] propose a replication scheme called symmetric replication. Each identifier is associated with k equivalent identifiers, where k is the suitable replication degree. That is, if N is the set of identifiers, N is partitioned into N/k classes. Each object is replicated at the nodes whose identifiers are equivalent to the root node's identifier. To preserve this invariant, when a new node joins the system, it cooperates with members of its class to obtain the list of objects to replicate. To access an object, one addresses the lookup request to any peer that stores the object replica, that is any node that belongs to the same class as the object root node.

Total Recall [2] is a peer-to-peer storage system that implements data availability as a first class storage property. It provides a means to characterize host availability. Total Recall proposes an architecture that permits to adapt the redundancy mechanism in function of the host availability characteristics. In practice, the system continuously monitors its constituent hosts, then derives the average host availability. Given a host availability level and a specified availability target, Total Recall provides insights on how to determine the redundancy degree that ensures with high probability that the data remains available. Total Recall permits to adapt the redundancy mechanism to the specifics of data and/or of the hosting infrastructure. Unlike Recall, *Pws* focuses only on ensuring replica availability.

III. System Overview

From the user point of view, *Pws* is a DHT-based storage system. Each replicated object is associated with a key that designates the group of replicas of that object. From the system point of view, *Pws* implements a flexible replicated object model, a multiple publication key replication approach and a neighborhood-based tracking of replica availability. It can serve as a replica maintenance substrate for peer-to-peer storage utilities destined to serve large size objects. An effort to develop such a system is presented in [13].

A. Flexible Replicated Object Model

Each object is associated with a replication contract that controls mainly the number of replicas to create and the replica loss repair strategy. A replication contract is enforced thanks to the cooperation of a group of managers. There is one manager group per replicated object.

Overall, within *Pws*, a replicated object is represented by two sets: R, the set of object replicas; and M, the set of replication managers that cooperate to enforce the replication contract associated with this object. Both object replicas and their associated replication managers are stored within the underlying storage overlay network.

 ISBN: 978-1-61208-280-6

The group of managers controlling a replicated object, as a whole, has the responsibility to maintain the suitable number of object replicas and of replication managers. These numbers may vary from one replicated object to another. It is worth noting that, for a given replicated object, its managers repair both object replica and manager loss.

B. Replica Placement

To replicate an object, *Pws* proceeds in three steps. Firstly, it determines the number of replicas to create and the number of managers that will be in charge of managing this replicated object, then attaches one distinguished local identifier to each replica and to each manager. Note that these decisions are guided by the replication contract associated with the object provider. A basic replication contract simply indicates the minimum number of replicas and of managers that the system has to maintain alive.

Secondly, *Pws* computes a distinguished key for each manager, then uses the computed key to place the corresponding manager. Upon the completion of each manager's installation, *Pws* notifies its identifier and its current location to the group of managers controlling the same replicated object.

Thirdly, once all managers of an object are placed, *Pws* places the object replicas. Again, for each replica, *Pws* computes a distinguished key, then uses it to place this replica. Once a replica is stored, *Pws* notifies its current location to managers controlling this replicated object.

Overall, upon the completion of the replication procedure, the suitable number of replicas and of managers are stored within the storage overlay network. Each manager of the new replicated object is aware of the current locations of all object replicas and of all managers.

Pws enforces final replica placements. That is, once a replica or a manger is placed at one node at the replication time, it will remain stored there as long as that node remains alive, regardless arrivals or departures of other storing nodes.

C. Neighborhood-based Peer's Availability Monitoring

To maintain the availability of a replicated object, *Pws* has to ensure that at anytime at least one replication manager and one replica remain available. *Pws* relies on watchers within each storing node's neighborhood to track the availability of both replicas and mangers stored at each storing node.

In practice, *Pws* associates with each storing node a set of nodes located within this node's neighborhood, called its *watch set*. *Pws* guarantees that, at each time, members of a node's watch set are all within this node neighborhood. This guarantees that when a node fails, its watchers are promptly informed by the network maintenance layer.

Each storing node advertises its contents to its watchers. An replica advertisement carries in particular the replica identifier together with the manager group identifier of the concerned replicated object. Thanks to these advertisements, a watcher can determine the contents lost when it is informed that a node that it monitors departed from the overlay network and can subsequently notify each loss to managers in charge of repairing this loss.

Overall, the neighborhood-based peer's availability monitoring permits to detect replica loss and to request its repair with no delay. This contributes to reduce the replica repair time which is a key metric that impacts data availability in a peer-to-peer storage system [2].

IV. Peer Watching System

The primary role of the peer watching system (Pws) is to maintain replica availability in presence of node churn. The key idea is to avoid replica migrations while enabling to detect replicas' loss with no delay and without paying for active replica tracking. For that, each storing node is associated with a set of nodes located within its neighborhood, called its *watch set*. Whenever a storing node leaves (voluntarily or involuntarily) the overlay network, its watchers repair its loss by notifying its departure to managers that control its stored contents.

A. Watch Set Initialization

A *Pws* node creates and initializes its watch set whenever a replica is first stored at this node since the last re-start. This is done in three steps.

Firstly, at the storing node, *Pws* requests the list of the node's replica set, thanks to an invocation to the primitive *replicaSet()* provided by the underlying KBR layer [8]. Once the replica set is retrieved, *Pws* selects at most $maxW$ elements to act as watchers of this storing node, where $maxW$ is a system configuration parameter, then sends a watch request message to each selected neighbor (see Function `createWatchSet()` of Figure 1). This message carries the identifier of the node to watch (i.e., the requester), the current value of the requester's `localTime`, and the set of selected watchers.

Secondly, at each selected watcher's node, upon the reception of a request to watch, *Pws* checks whether the local node has been watching this requester in the past (see Function `onReceiveRequestToWatch()` of Figure 1). If the receiver holds none information, it creates a new watched node descriptor, passing to the constructor the information contained in the watch request message. If however the receiver node does have a descriptor representing the requester within its `watchedNodes` structure, it updates that descriptor. Note that, this could happen if the receiver was a member of the requester's watch set, then at some point was pushed out the requester's neighborhood. Upon the completion of the treatment of the request to watch a server, the receiver acknowledged the highest timestamps received so far.

Finally, at the requester storing node, upon the reception of a response from each candidate watcher, *Pws* updates the structure that describes the corresponding watcher. In particular, for each watcher which is aware of replicas stored by this replica server, it registers the timestamps of the most recent received advertisement.

 ISBN: 978-1-61208-280-6

B. Watch Set Maintenance Protocol

The objective of the watch set maintenance protocol is to make sure that for each storing node, there always exists at least one watcher within this storing node's neighborhood. Though initially each watcher is peaked from the neighborhood of the storing node that it is watching, arrivals of new nodes can push it out this neighborhood. Also, watchers can leave (voluntarily or involuntarily) the overlay network.

Function `onJoinOrLeave()` of Figure 1 sketches the watch set maintenance procedure. The watch set maintenance procedure exploits update upcalls from the underlying overlay network maintenance facility. Precisely, when a change occurs in the local node neighborhood, an update upcall is issued to this node, passing it the identifier of the node that departed or joined [8]. The *Pws* provided `update()` then invokes the watch set maintenance procedure which behaves as follows. Firstly, *Pws* retrieves the list of neighbors of the local node then computes, w, the number of alive watchers that are still in the local node's neighborhood. If $w \leq wThreshold$, *Pws* selects $maxW - w$ neighbors that are not currently watching the local node and adds them to the local node's watch set. Secondly, *Pws* notifies the current watch set membership to watch set members.

C. Advertisement Protocol

The objective of this protocol is to make sure that, whenever a storing node leaves the storage overlay network, its watchers are aware of replicas stored there and of managers that can repair their losses. The problem arises from the fact that the contents of each storing node change over time: new replicas can be added and old replica can be deleted. The challenge is to conciliate conflicting requirements: maintaining the network bandwidth consumption as low as possible while ensuring that in case of the departure of a storing node, that node's watchers have an accurate list of replicas that are lost.

Periodically and after each reconfiguration of the watch set of a node, *Pws* requests the watching service to advertise the local node's contents to its watch set, thanks to the execution of Function `onRequestToAdvertise()` of Figure 2 that permits to advertise recently stored contents to members of the local node's watch set. An advertisement message contains mainly : the node identifier, the current local time, and the descriptors of replicas stored since the most recent advertisement that has been already acknowledged by all watchers. Each advertised replica descriptor comprises this replica local identifier, the group identifier of its managers, and its storage time.

Function `onAdvertisementDelivery()` of Figure 2 sketches actions performed by each watcher upon the reception of an advertisement. Firstly, it updates accordingly its view of the watched node. In particular, each watcher registers the descriptors of replicas that it was not yet aware of; it also updates the most recent timestamps received from the current sender. Then, upon the completion of its treatment, it acknowledges the reception and the treatment of the advertisement.

```
// Data structures maintained by each PWS node
thisNode; // Reference to the local node
watchSet; // Set of watchers of this node
wThreshold; // Minimum number of watchers per node
maxW; // Maximum number of watchers per node.
cts; // Current timestamps of the local storing node.
watchedNodes; // PWS nodes watched by this node

function createWatchSet()
Node[] replicaSet = thisNode.replicaSet();
if replicaSet.isEmpty() then
    return 0;
end if
for i = 1 to i = maxW do
    watcher = new Watcher(replicaSet[i]);
    watchSet.add(watcher);
end for
cts = initTimestamp();
notifyWatchers();
return watchSet.size();
end function

function notifyWatchers()
watchers = watchSet.getWatchers();
req = new RequestToWatch(thisNode.id, watchers , cts);
for all w ∈ watchers do
    route(w.getNodeId(), req, null);
end for
nbResponses = 0;
while nbResponses < watchers.size() do
    waitNextResponse2RequestToWatch();
    r = getNextResponse2RequestToWatch();
    watchSet.updateLastRcvTS(r.watcher, r.lastRcvTS);
    nbResponses++;
end while
end function

function onReceiveRequestToWatch(req)
n = req.requester;
if (watchedNodes == ∅)||(n ∉ watchedNodes) then
    watchedNode = new WatchedNodeDesc(n)
    watchedNodes.add(watchedNode);
end if
watchedNodes.updateInfo(n, req);
lastRcvTS = watchedNodes.getLastRcvTS(n);
r = new Response2RequestToWatch(thiNode.id, lastRcvTS);
send(n, r);
end function

function onNeighborJoinLeaveNotification()
Node[] neighbors = thisNode.neighborSet();
currentWatchers = watchSet.getWatchers();
for all w ∈ currentWatchers do
    if w ∉ neighbors then
        watchSet.remove(w);
        sz = watchSet.size();
    end if
end for
if (sz > wThreshold) then
    return
end if
replicaSet = thisNode.replicaSet();
while ((0 < i < maxW) && (sz < maxW)) do
    if (replicaSet[i] ∉ watchSet) then
        watchSet.add(replicaSet[i]);
        sz++;
    end if
    i++;
end while
notifyWatchers();
end function
```

Fig. 1. WatchSet membership maintenance

At the storing node side, upon the reception of a response from a watcher, *Pws* updates the corresponding state maintains within the `watchSet` data structure.

 ISBN: 978-1-61208-280-6

D. Replica Loss Detection and Notification

Given the set of watchers of a node, we define the primary watcher for this node to be the watcher with the smallest node identifier, bigger or equal to this node's identifier. When a node detects the departure of a storing node that it is watching, it first determines whether it is the primary or a secondary watcher of this storing node. As watch sets change over time and since changes are not notified atomically to watch sets' members, the watch set membership view can differ from one member to another. Hence, two distinct watchers can decide different primary watchers. This will result in more notifications than necessary.

If a watcher decides to be the primary watcher, it notifies the loss of each advertised replica to the advertised managers. Otherwise, it sends a request to participate to notifications to the watcher that it considers to be the primary. Such a request contains in particular the identifier of the node that has failed.

Upon the reception of a request to participate to notifications, the (presumably primary) watcher checks if it has already detected the notified departure and if it has decided to be a primary watcher of the failed node. If both conditions are not satisfied, it promotes itself a primary watcher for the departed node, then acts accordingly. Once notifications have been sent to suitable managers, it sends back a response to the requesting secondary watcher. This response contains the timestamps of the most recent replica advertised to it by the failed storing node.

Upon the reception of a response from a primary watcher, a secondary watcher checks it it were advertised replicas that its primary is missing. If any, it notifies their loss to their advertised managers. If however a secondary watcher receives none response to its request to participate to notifications, after some delay, it promotes itself a primary and acts accordingly.

Regardless which watcher (primary or secondary) does it, a replica loss notification contains the lost replica's identifier, its location, and its creation timestamps.

E. Replica Maintenance Protocols

Pws ensures data replica availability thanks to the cooperation between watchers in the neighborhood of storing nodes and managers of replicated objects, also replicated towards the overlay network. Watchers are informed with no delay of replica or manager losses, thanks to update upcalls issued by the overlay network maintenance layer. Once informed, watchers cooperate with one another to notify appropriate managers. Managers, in turn, guarantee that at anytime, at least one data replica and one manager remain available.

Pws balances the handling of replica losses among managers controlling each replicated data. In practice, the loss of a replica (or manager) is handled by the manager with the smallest local identifier among managers that remain alive, equal to or greater than the lost replica's local identifier. Each manager maintains the list of its peers that are still alive, and the list of outstanding repair requests.

Pws enforces two separate replica maintenance protocols: one for replication managers and one for object replicas.

```
// Additional structure maintained by each PWS node
storedReplicas; // Set of descriptors of replicas stored locally

function onRequestToAdvertise()
Timestamps lbackts = watchSet.computeLowerBoundAcknowledgedTS();
if lbackts < cts then
    adv = new Advertisement(thisNode.id, cts);
    for all r ∈ storedReplicas do
        if (r.storatgeTime > lbackts) then
            adv.addReplica(r.id, r.manager, r.storageTime);
        end if
    end for
    for all w ∈ watchSet do
        route(w.getNodeId(), adv, null);
    end for
end if
end function

function onAdvertisementDelivery(adv)
TimeStamp lastRcvTs = watchedNodes[adv.notifier].lastRcvTs;
if (adv.cts < lastRcvTs) then
    return lastRcvTs
end if
for all replica ∈ adv do
    if (replica.storageTime > lastRcvTs) then
        watchedNodes[adv.notifier].registerReplica(replica);
    end if
end for
```

Fig. 2. Replica advertisement related procedures

1) Manager Maintenance Protocol: When a manager is notified the loss of another manager controlling the same replicated data, firstly, it determines the identifier of the primary manager that will lead the reparation. This is a symmetric election process that exploits the list of alive peers maintained by each manager.

Secondly, it marks the lost manager as unavailable and adds the corresponding manager repair request descriptor within the local list of outstanding repair requests.

Thirdly, the manager that leads this reparation, allocates a local identifier for a new manager, computes the key k_{new} that identifies this new manager, then routes a request to deploy a manager with key k_{new} towards the overlay network of storage nodes. This request contains the list of existing replicas and managers. Upon its installation, the new manager notifies its arrival to existing managers controlling the same replicated object.

Upon the notification of the arrival of the new manager in replacement of a failed one, each manager removes the corresponding repair request from the list of outstanding requests to repair, then adds the new manager within the list of alive managers.

Finally, once the reparation is terminated, one re-examines the list of outstanding requests and re-processes any outstanding request for which the lost manager was the leader.

2) Data Replica Maintenance Protocol: When a manager is notified the loss of a data replica, it first adds the corresponding replica repair request within the list of outstanding repair requests. Then, this manager checks whether it is the leader for this replica loss?

The manager responsible of the lost replica allocates a new local replica identifier, computes a new replica key, then routes the request to create a replica corresponding to this key towards the overlay network of storage. To create a new

replica a node, one can simply get a copy of an existing replica. Once the new replica is created, *Pws* notifies its location to each manager in charge of that replicated data.

Upon the reception of the new replica location, each manager updates its list of replica locations, then removes the corresponding request to repair.

V. Analytical Evaluation

The objective is to compare the cost incurred by *Pws* with the one incurred by a basic leaf set-based replica maintenance system (BLS) to enforce replica availability in presence of node churn. We consider two metrics: the number of messages exchanged over the network and the network bandwidth consumed to enforce data availability. We consider a storage overlay that has already reached its equilibrium. We restrict the problem to the one of enforcing the availability of a set of already replicated objects. That is, none request to replicate a new object is issued any more.

The number of watchers enforced by *Pws* for each storing node is equal to the replication degree for both *Pws* and the leaf-set based replica maintenance system. Consequently, *Pws* maintains as many object replicas as replication managers per replicated object. Furthermore, we assume that *Pws* spreads out replicas and managers of each replicated object over distinct storing nodes.

Let consider the following notations: ω, the replication degree; γ the average total number of replicas (and hence of managers) stored at each node; μ, the average size of each object replica; and η the size of meta data maintained by a manager. Finally, let $\lambda = max(\eta, \tau_1, \tau_2, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5)$, where: τ_1 is the size of the descriptor of a watcher; τ_2, the size of the descriptor of a replica as perceived by a watcher; ϕ_1, the size of the response returned by a watcher on the reception of new membership list; ϕ_2, the size of the response returned by a watcher on the reception of an advertisement; ϕ_3, the size of the request to participate to notifications of replica losses; ϕ_4, the size of the response to a request to participate to notifications; ϕ_5, the size of each replica loss notification. Note that these parameters are implementation-dependent.

A. Overhead on New Node Arrival

a) Pws Cost: When a new node joins the overlay network, this arrival impacts the watch set membership of the ω closest neighbors of the new node. Hence, in the worse case *Pws* runs a watch set maintenance procedure on each of the ω closest neighbors of the new node. For each concerned neighbor, *Pws* constitutes a watch set membership, then this new watch set to its members. This amounts to ω update requests (each request carries the new watch set membership), plus the same number of responses. In addition, on each watch set update, the watched node advertises its stored contents to its new watchers. With our assumption, at worse, there is one new watcher for each of the ω closest neighbors.

If $t4join_{pws}$ (resp. $bc4join_{pws}$) denotes the number of network messages (resp. number of bytes) transmitted over the network by *Pws* on each node arrival to enforce data availability, the following equations hold.

$$t4join_{pws} \leq 2\omega(\omega + 1) \quad (1)$$

$$bc4join_{pws} \leq \lambda(\omega^3 + \omega^2 + \omega + \gamma) \quad (2)$$

b) BLS Cost: With a basic leaf set-based replica maintenance, whenever a new node joins the overlay, it cooperates with its ω closest neighbors to determine the objects to replicate. Firstly, each of the ω closest neighbors of the new node computes the list of object identifiers that the new node should replicate, then sends this list to the new node. Each list contains in average γ/ω object identifiers. Later on, the new node will request each of its neighbors to send it each object that it should replicate. Once the new node has complete replica migrations, it informs each of its neighbors. If $t4join_{bls}$ (resp. $bc4join_{bls}$) denotes the traffic (resp. bandwidth consumed) on a node arrival by a leaf set-based replica maintenance system, the following equations hold:

$$t4joinbls \geq \omega + \gamma \quad (3)$$

$$bc4join_{bls} \geq \gamma\mu \quad (4)$$

c) Comparison: From [1] and [3], we observe that on node arrival, *Pws* will generate less traffic BLS provided that $\gamma > 2\omega^2 + \omega$. With respect to network bandwidth consumption, let r_j denote the ratio $bc4join_{pws}/bc4join_{bls}$; from [2] and [4], we derive $r_j \leq \lambda\mu^{-1}[1 + (\omega^3 + \omega^2 + \omega)\gamma^{-1}]$. This ratio indicates that the larger μ compared to λ, the better *Pws* compared to BLS. To illustrate, consider a prototype implementation that enforces $\lambda \leq 10^3$ and let $\omega = 4$ and $\gamma = 50$. If μ is equal to 10^6, then on network arrival BLS consumes 373 times more network bandwidth than *Pws*. It is worth noting that in [4] we consider only the bandwidth due to replica migrations.

B. Overhead on Node Departure

d) Pws Cost: When a node departs from the underlying network overlay, the underlying network management layer notifies this departure to nodes in its neighborhood, and hence to each member of its watch set. Upon a departure, in addition of the overhead (equivalent to the one incurred in case of a new node arrival) due to the watch set maintenance procedure, *Pws* incurs the following additional costs:

- Inter watchers cooperation. Watch set members cooperate with one another to determine which one is responsible to notify object losses to their suitable managers. This consists on $\omega - 1$ requests issued by secondary watchers and as much responses from the primary watcher (for more detail on inter watchers cooperation protocol, refer to Section IV-D).
- Loss notifications. For each entity stored at the departed node, one loss notification is sent to its manager. In average, there are 2γ loss notifications sent over the network.

 ISBN: 978-1-61208-280-6

- Loss repairs. To repair a replica (resp. replication manager) loss, its manager issues a request to create a new replica (resp. replication manager) at a random storing node. In average, on each node departure, there are γ requests to recreate replicas and γ requests to recreate replication managers. Note that each request to create a replica contains a means to help locate existing replicas, while a manager creation request carries the whole management state.
- Replica and manager recreations. To recreate a new replica, *Pws* requires 2 messages: one request to an existing replica to retrieve its state and a response containing the replicated state. Note that, to create a new manager, there is no need of network communication. Also, each newly created replica or manager is notified to the group of managers of its replicated object.

Overall, if $t4depart_{pws}$ (resp. $bc4depart_{pws}$) denotes the number of messages (resp. bytes) transmitted over the network by *Pws* on each node departure to enforce data availability, the following equations hold.

$$t4depart_{pws} \leq 2\omega^2 + 4\omega + 8\gamma \quad (5)$$

$$bc4depart_{pws} \leq \lambda[\omega^3 + \omega^2 + 2\omega + 8\gamma] + \mu\gamma \quad (6)$$

e) BLS Cost: With a basic leaf set-based replica maintenance system, whenever a node departs from the overlay network, a replica maintenance procedure is run by its ω closest neighbors in order to recreate replicas that are lost due to this departure. Let m designate one of the ω closest neighbors of the departed node. Firstly, the system computes the list of objects stored by the departed node for which m is (currently) the root. Secondly, for each such object, the system peaks one of the ω closest neighbors of m that doesn't yet replicates this object and sends it a request to create a replica.

On average, each m requests the recreation of $\gamma/\omega)$ replicas that were stored at the departed node. To recreate each replica, at least one request of the object state and one response containing the requested state are requires. Hence, if $t4depart_{bls}$ (resp. $bc4depart_{bls}$) denotes the number of messages (resp. bytes) transmitted over the network by a leaf set-based replica maintenance system, the following equations hold:

$$t4depart_{bls} \geq \omega + 2\gamma \quad (7)$$

$$bc4depart_{bls} \geq \gamma\mu \quad (8)$$

f) Comparison: On node departure, *Pws* incurs more traffic and consumes more network bandwidth to recreate managers and data replicas that are lost. From [6] and [8] it comes that if we consider large size objects (e.g., $\mu \geq 10^8$) both systems incurs comparable overhead on node departure. To see why, one could observe that as the average size of stored contents, the number of objects per node tends to diminish and so is the replication degree. We anticipate that, for large size objects both $\lambda\gamma$ and to $\lambda\omega^3$ are negligible compared to μ.

VI. Conclusion

We presented *Pws*, a peer-to-pee watching system that constitutes a replica maintenance substrate for DHT-based storage network. *Pws* that enforces a multiple publication keys replication approach, while avoiding an active tracking of storing nodes availability. We introduced the notion of node's watch set and detailed how to guarantee that watch set members remain in the neighborhood of the watched node in presence of node churn. We also presented the cooperation between storing nodes and watchers on the one hand, and between watchers and mangers on the other hand. The analytical evaluation of *Pws* overhead confirms that this system is an interesting alternative to maintain data availability for peer-to-peer storage utilities that destined to serve large-size objects.

References

[1] K. Kyungbaek and P. Daeyeon, "Reducing data replication overhead in dht based peer-to-peer system," in *Proceedings of the 2006 International Conference on High Performance Computing and Communications*, vol. 4208. LNCS, 2006, pp. 915 – 924.

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker, "Total recall: system support for automated availability management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*. Berkeley, CA, USA: USENIX Association, 2004, pp. 337–350.

[3] K. Tati and G. M. Voelker, "On object maintenance in peer-to-peer systems," in *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[5] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 41–53, 2004.

[6] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, ser. Middleware '01. London, UK, UK: Springer-Verlag, 2001, pp. 329–350.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 161–172, Aug. 2001.

[8] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," *IPTPS03 International workshop on PeerToPeer Systems*, pp. 33–44, 2003.

[9] P. Druschel and A. Rowstron, "Past: A large-scale, persistent peer-to-peer storage utility," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE, 2001, pp. 75–80.

[10] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 188–201.

[11] S. Legtchenko, S. Monnet, P. Sens, and G. Muller, "Churn-resilient replication strategy for peer-to-peer distributed hash-tables," in *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 485–499.

[12] A. Ghodsi, L. O. Alima, and S. Haridi, "Symmetric replication for structured peer-to-peer systems," in *Proceedings of The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, Trondheim, Norway, 2005, p. 12.

[13] M. Makpangou, "P2p based hosting system for scalable replicated databases," in *Proceedings of the 2009 EDBT/ICDT Workshops*, ser. EDBT/ICDT '09. New York, NY, USA: ACM, 2009, pp. 47–54.

 ISBN: 978-1-61208-280-6